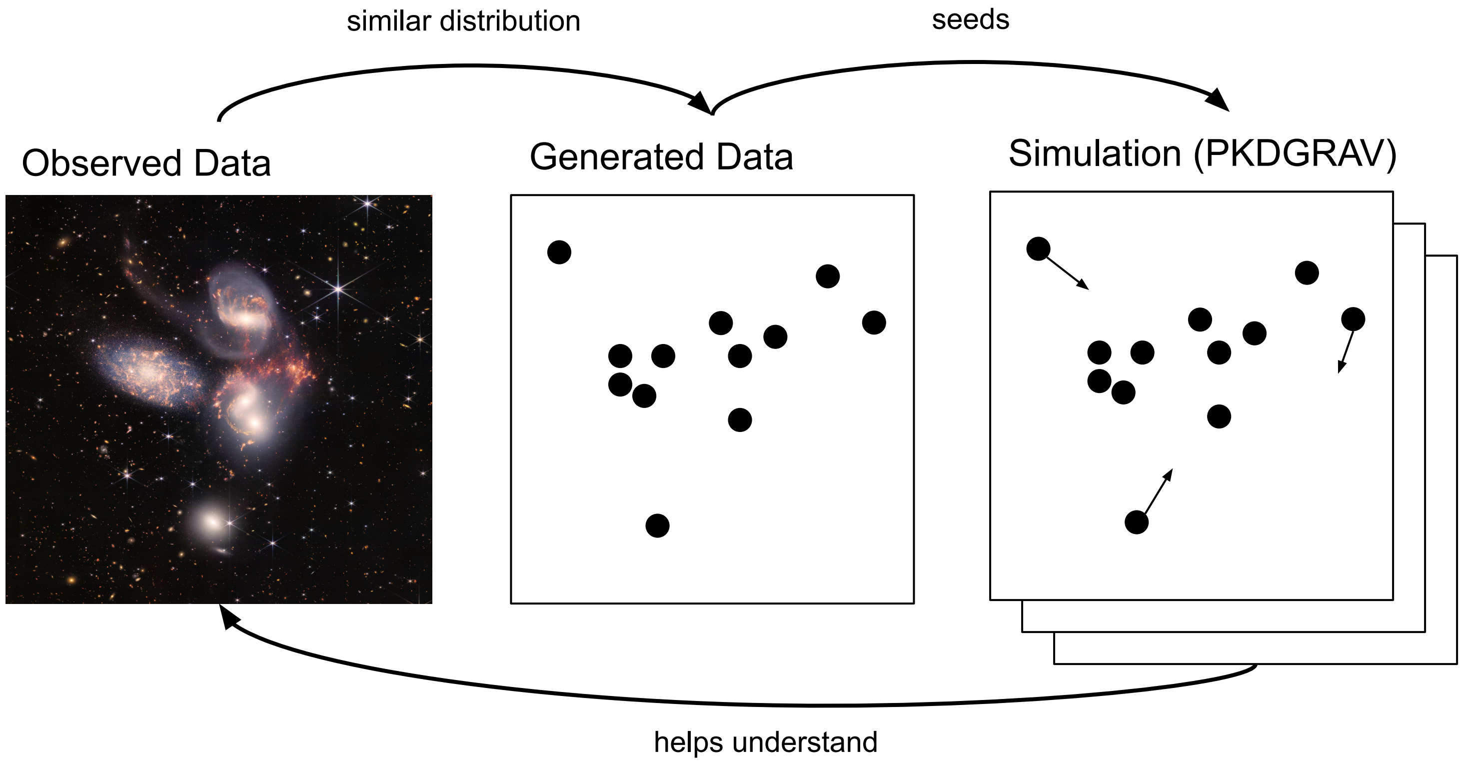
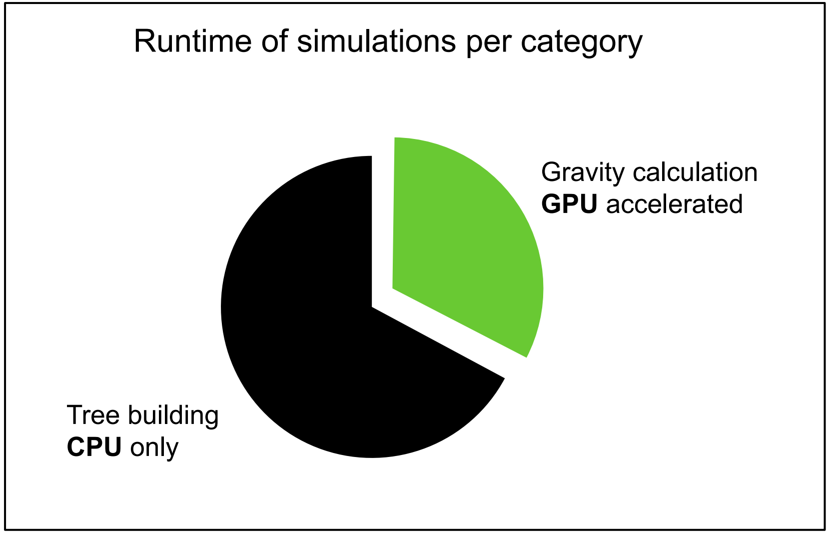
Orthogonal Recursive Bisection on the GPU for Accelerated Load Balancing in Large N-Body Simulations

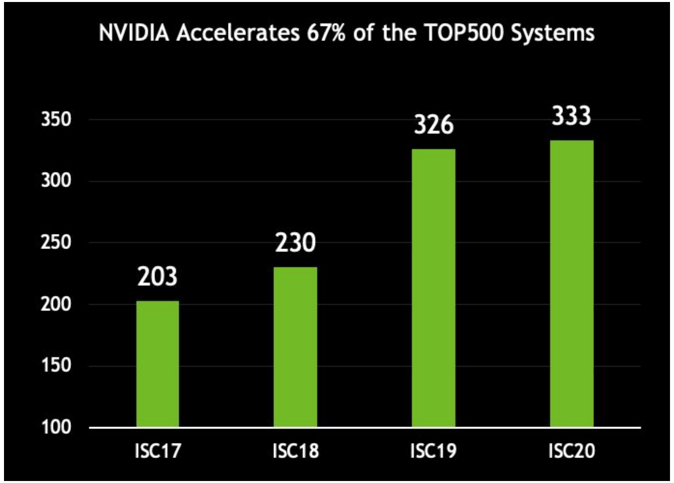
Bachelor Thesis - Andrin Rehmann
UZH 2022

Astrophysical Simulations



Motivation:

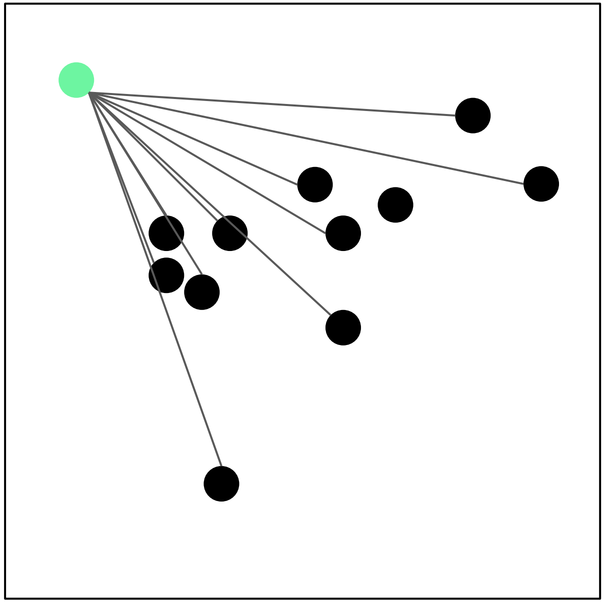




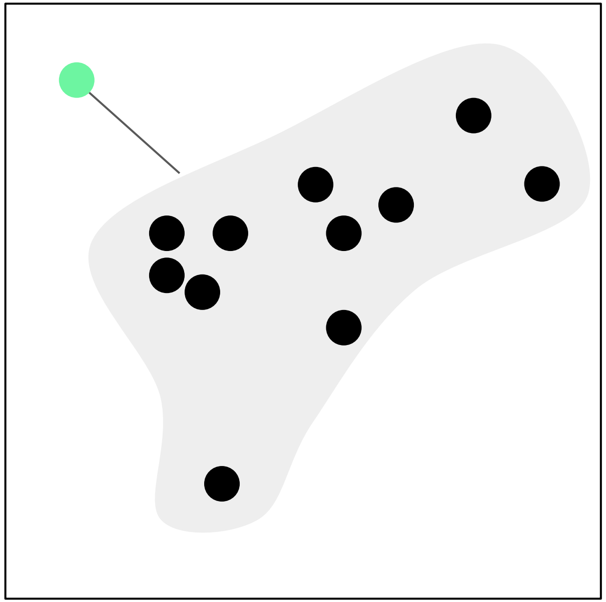
As of 2020, more than ⅔ of all supercomputers are CUDA enabled

Fast Multipole Method (FMM)

Naive brute force method **O(N^2)**

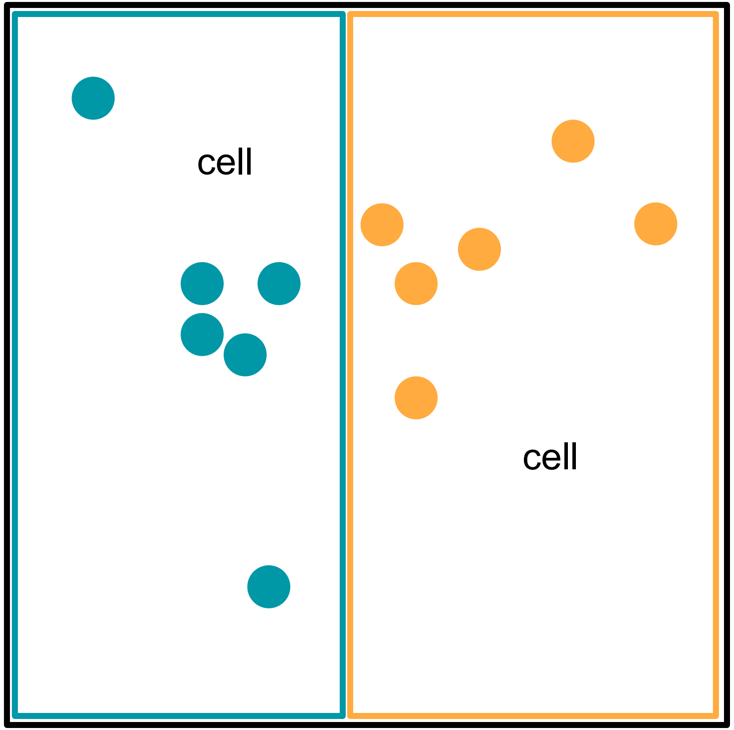


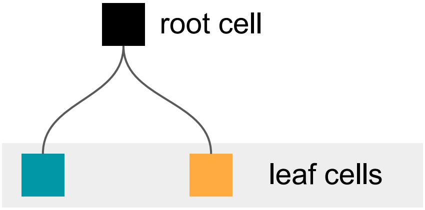
Fast multipole method **O(N)**



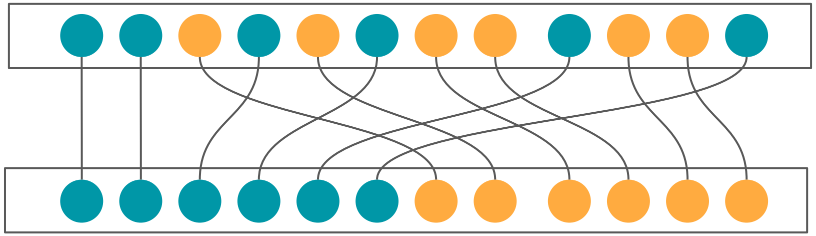
The grouping of particles is done with the **ORB** algorithm

Orthogonal Recursive Bisection **ORB**

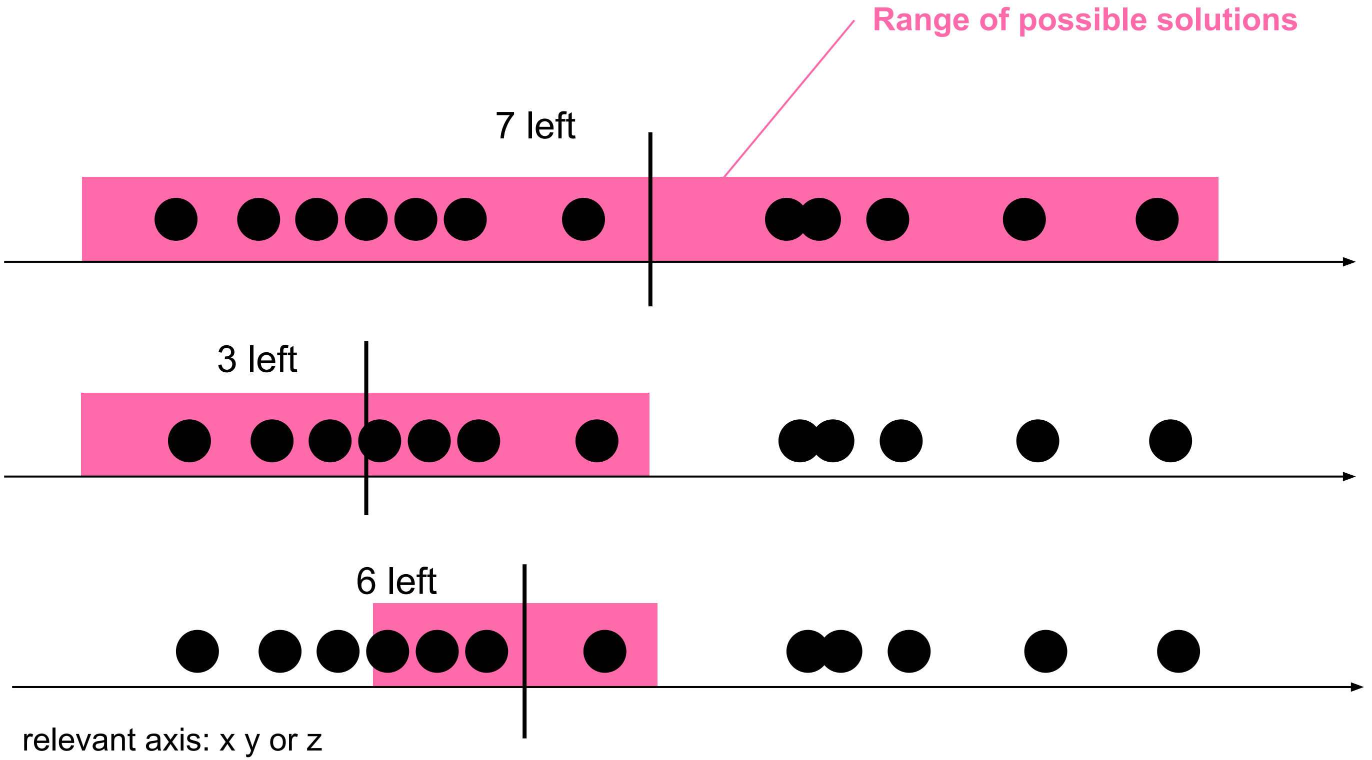




Partition particles for fast access

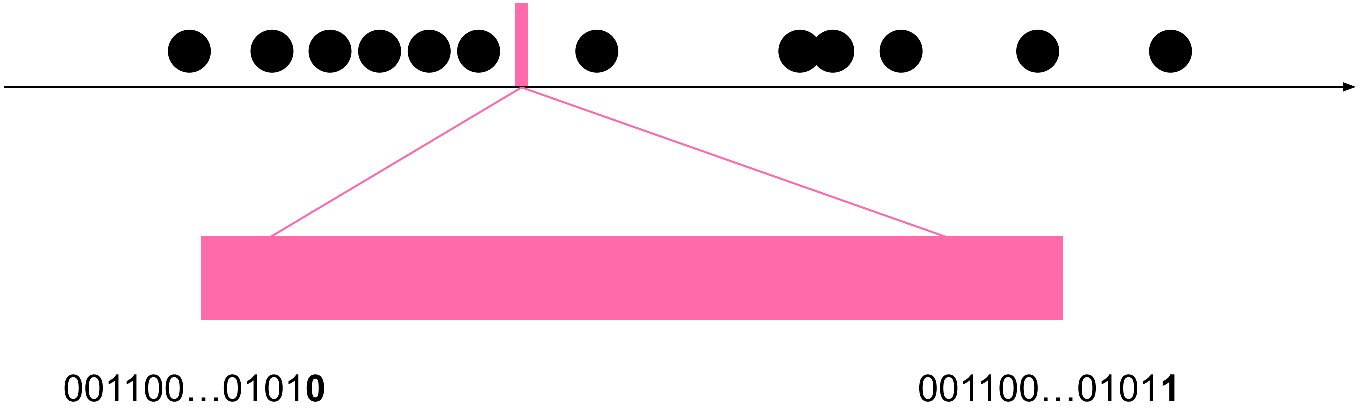


Bisection method

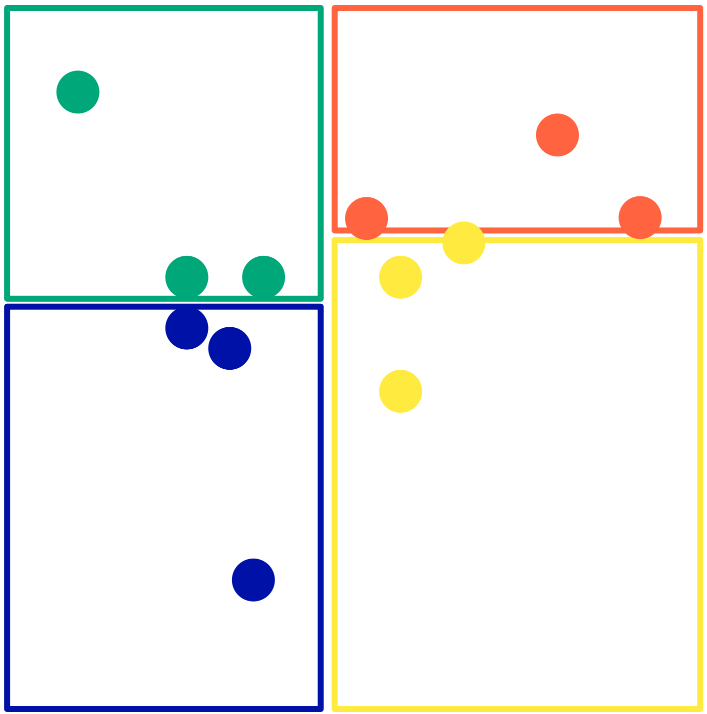


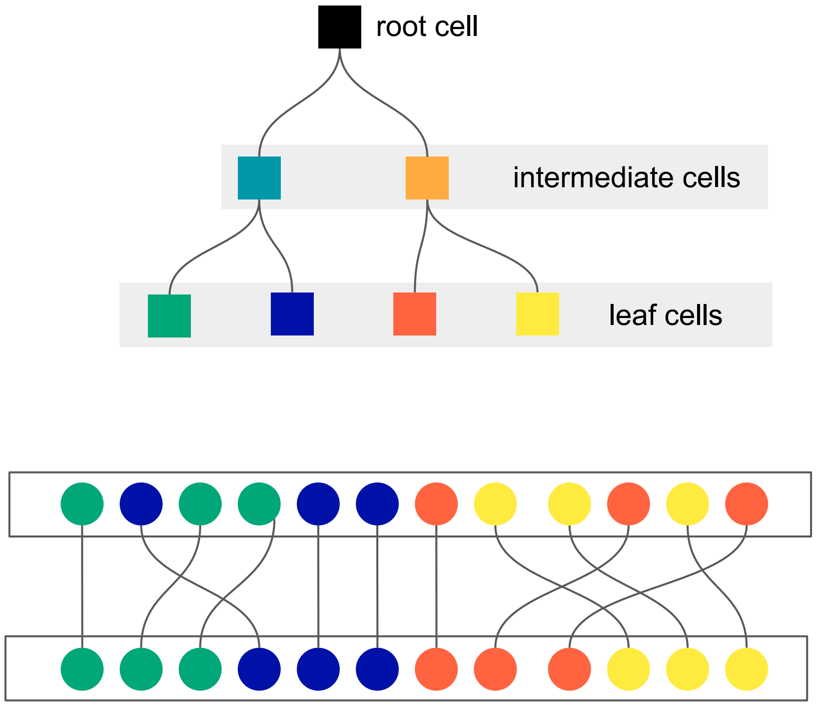
We aim for 12 / 2 = **6** particles smaller or left of the cut

O(32) runtime for 32 bit precision



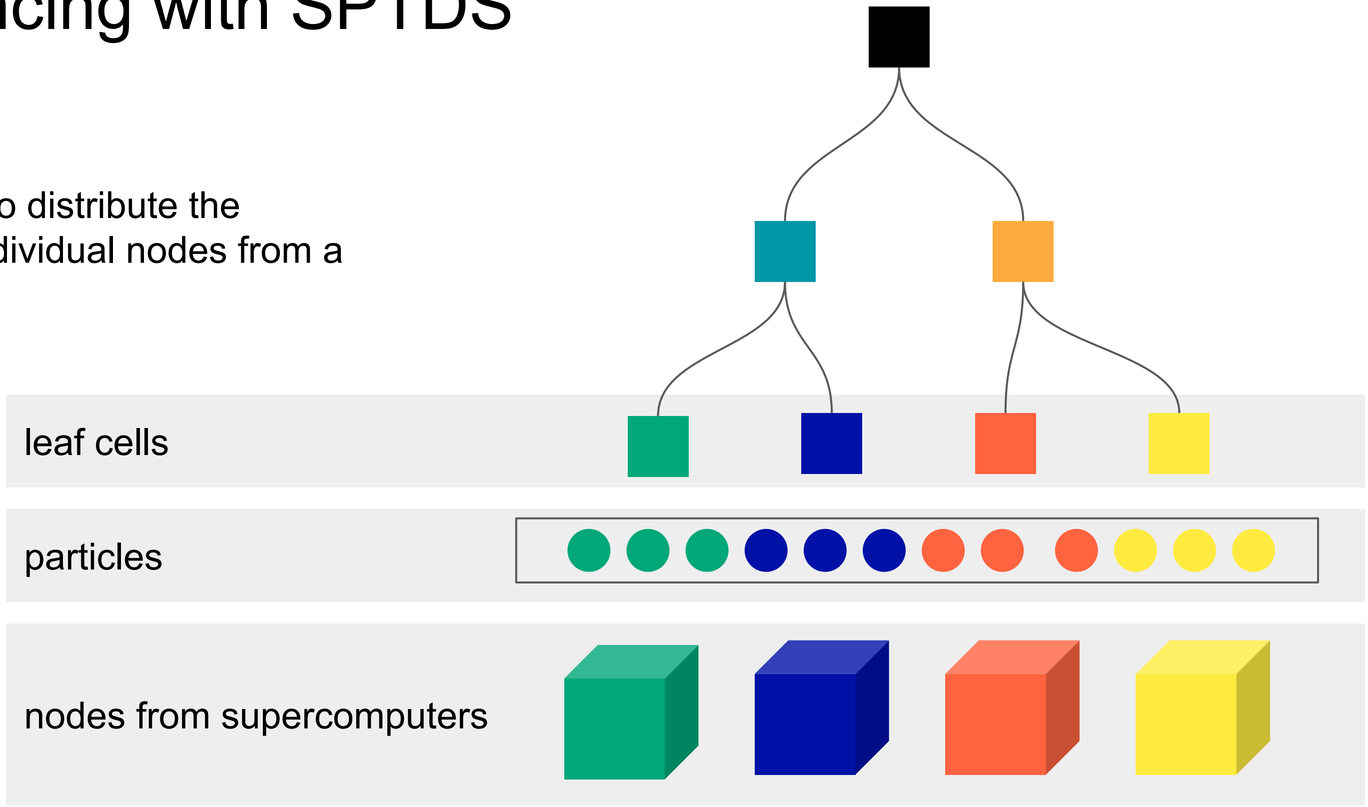
Space Partitioning Tree Data Structure (SPTDS)



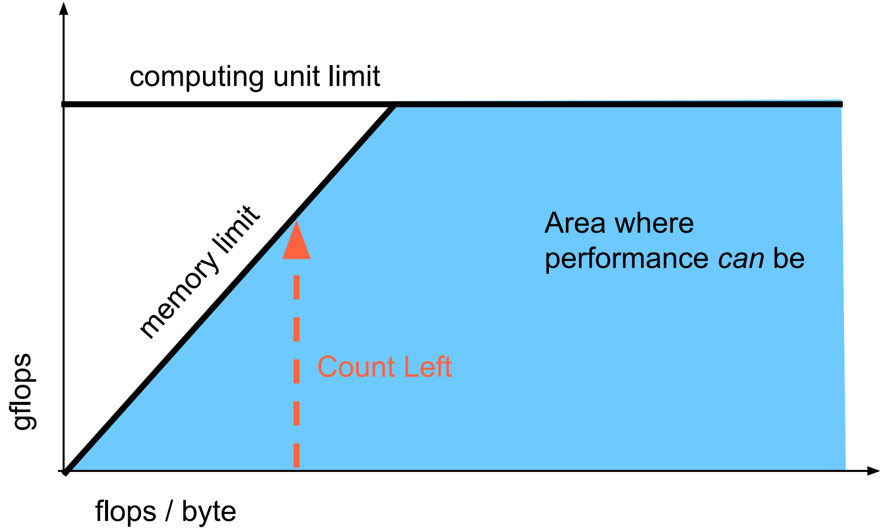


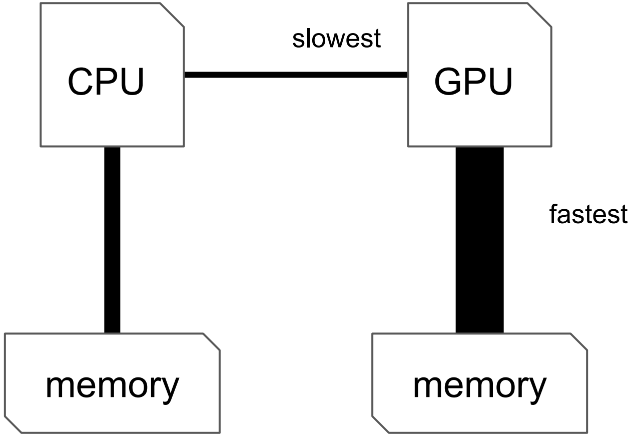
Load Balancing with SPTDS

Tree can be used to distribute the particles among individual nodes from a supercomputer



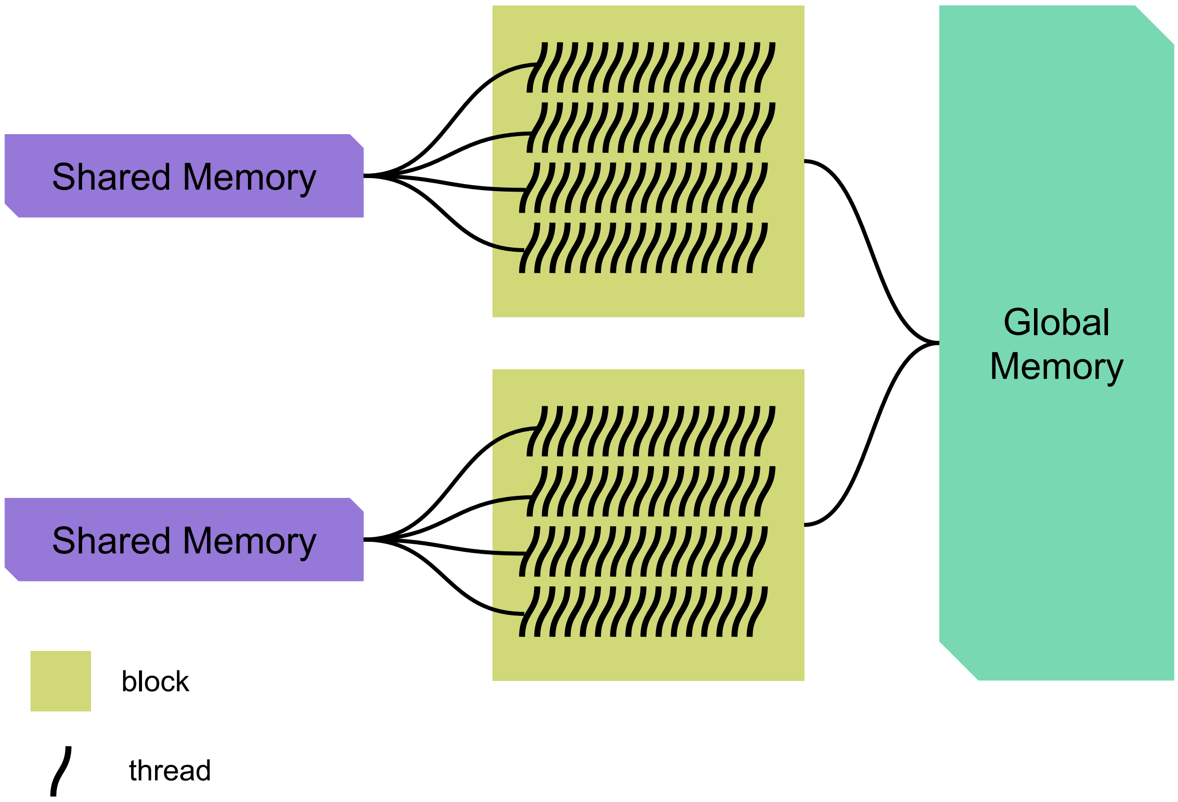
6.2x speedup of GPU over CPU version expected





1. Use GPU as **much** as possible
2. Use data link between CPU and GPU as **little** as possible

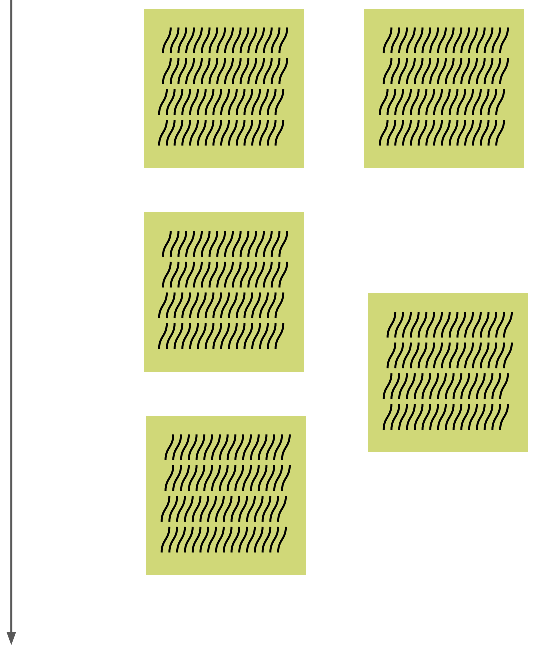
CUDA



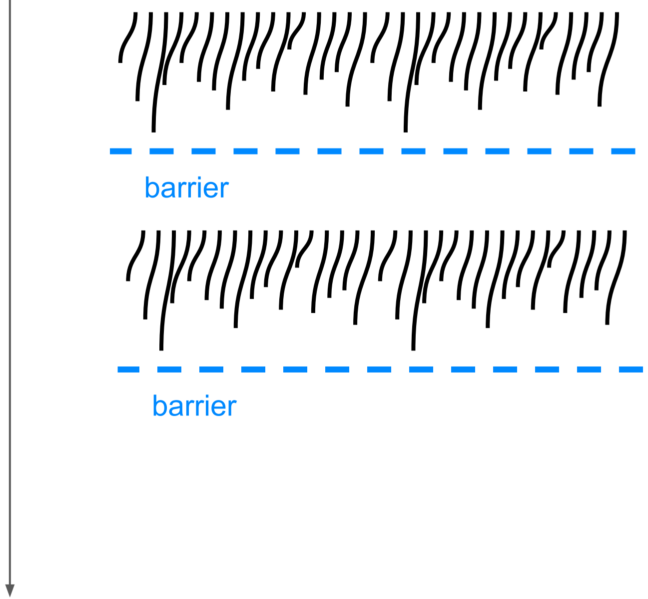
- thousands of blocks can exist
- up to 1024 threads per block
- only threads from same block can access a shared memory register
- all threads can access the global Memory

Synchronization

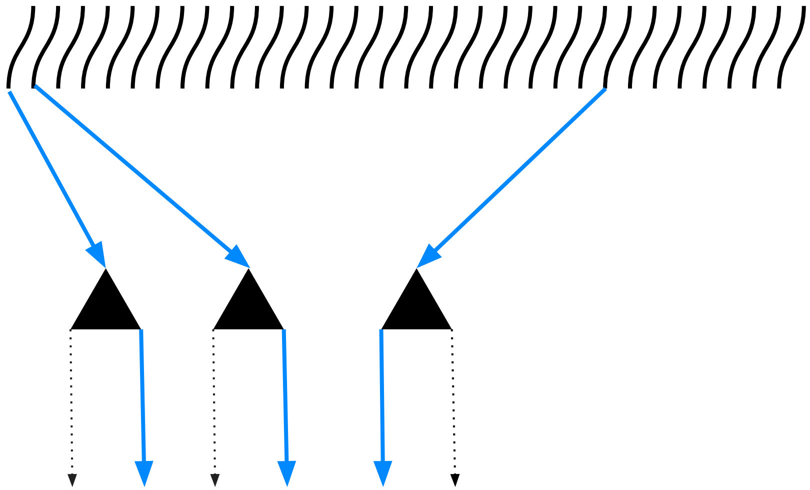
blocks within warp cannot be synchronized



threads within block can be synchronized



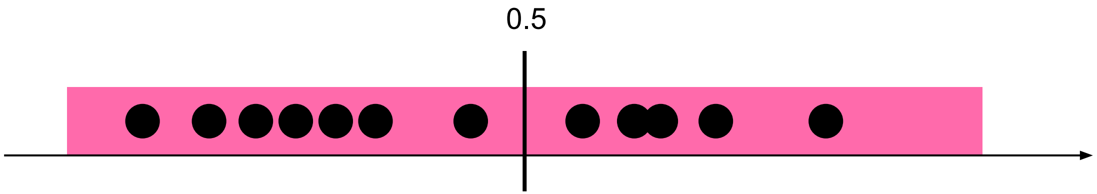
Warps



branch divergence: thread executes
different code from other threads in warp

- ➔ A warp consists of 32 threads
- ➔ Synchronization is possible and needed to avoid data race
- ➔ Warp level primitives can be used to implement most algorithms on thread level

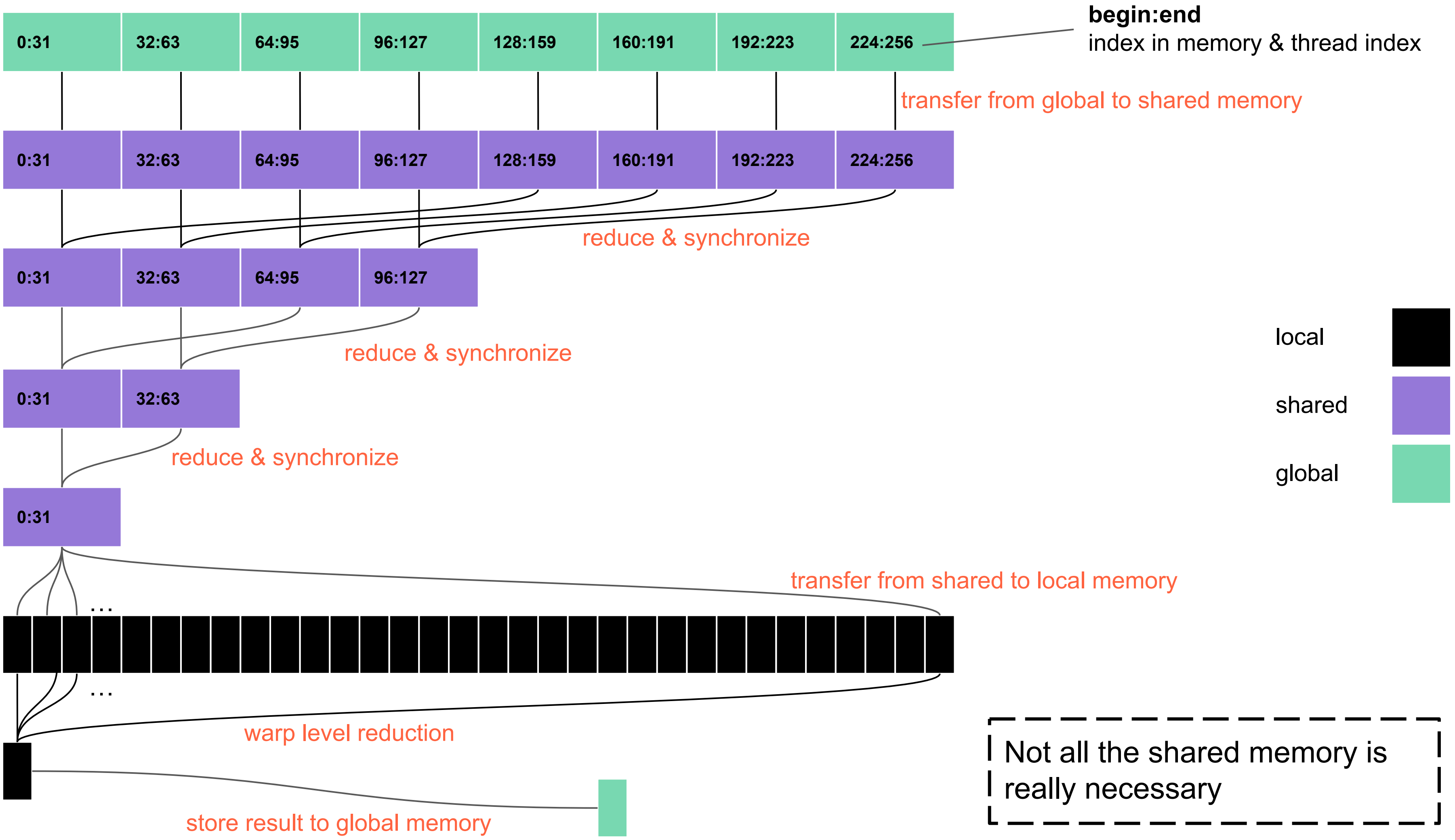
Bisection method



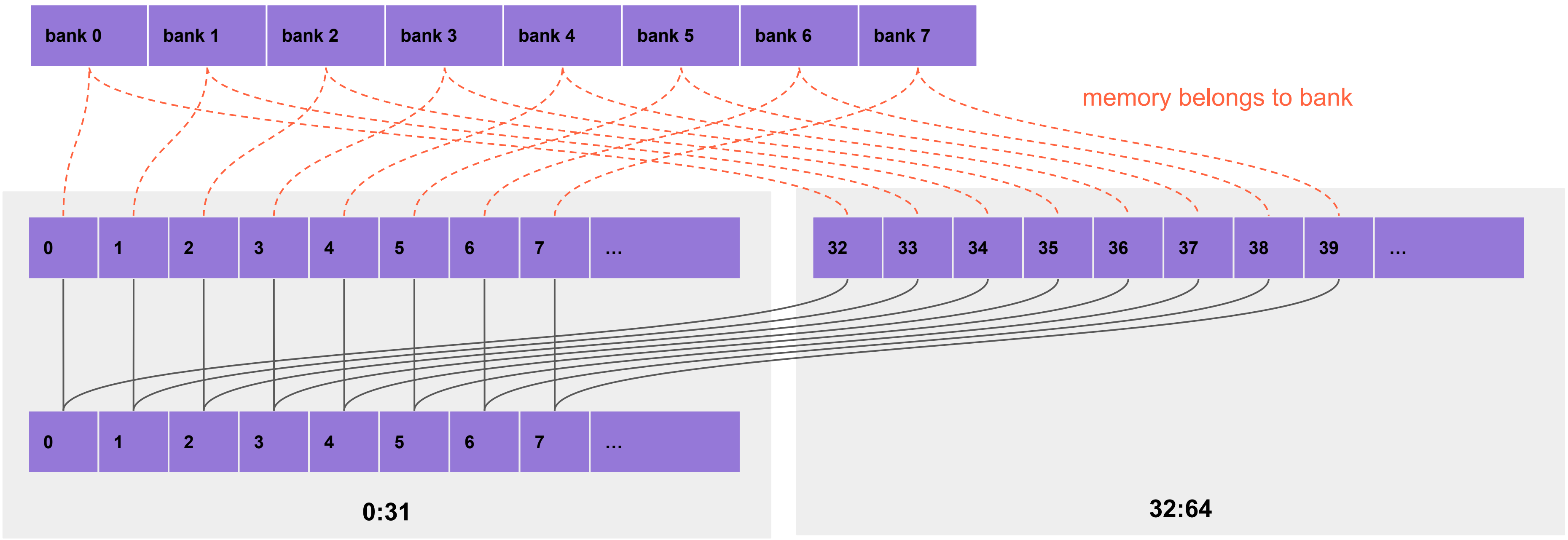
0.9 < 0.5 + 0.1 < 0.5 + 0.3 < 0.5 + 0.6 < 0.5 + 0.7 < 0.5 + …

(0.9 < 0.5 + 0.1 < 0.5) + (0.3 < 0.5 + 0.6 < 0.5) + (0.7 < 0.5 + …

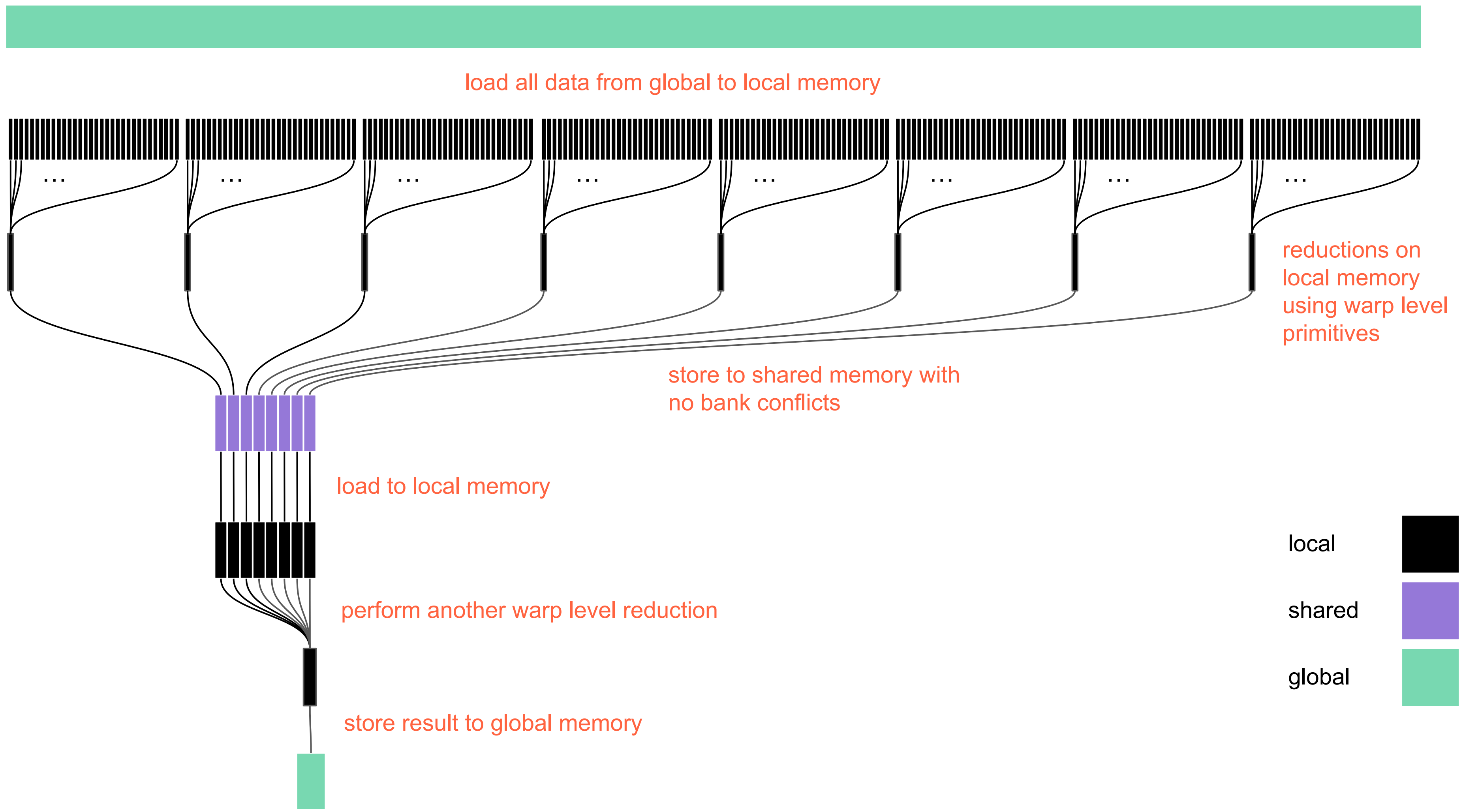
We can easily split the Count Left part of the bisection into multiple sub problems: ideal for CUDA.

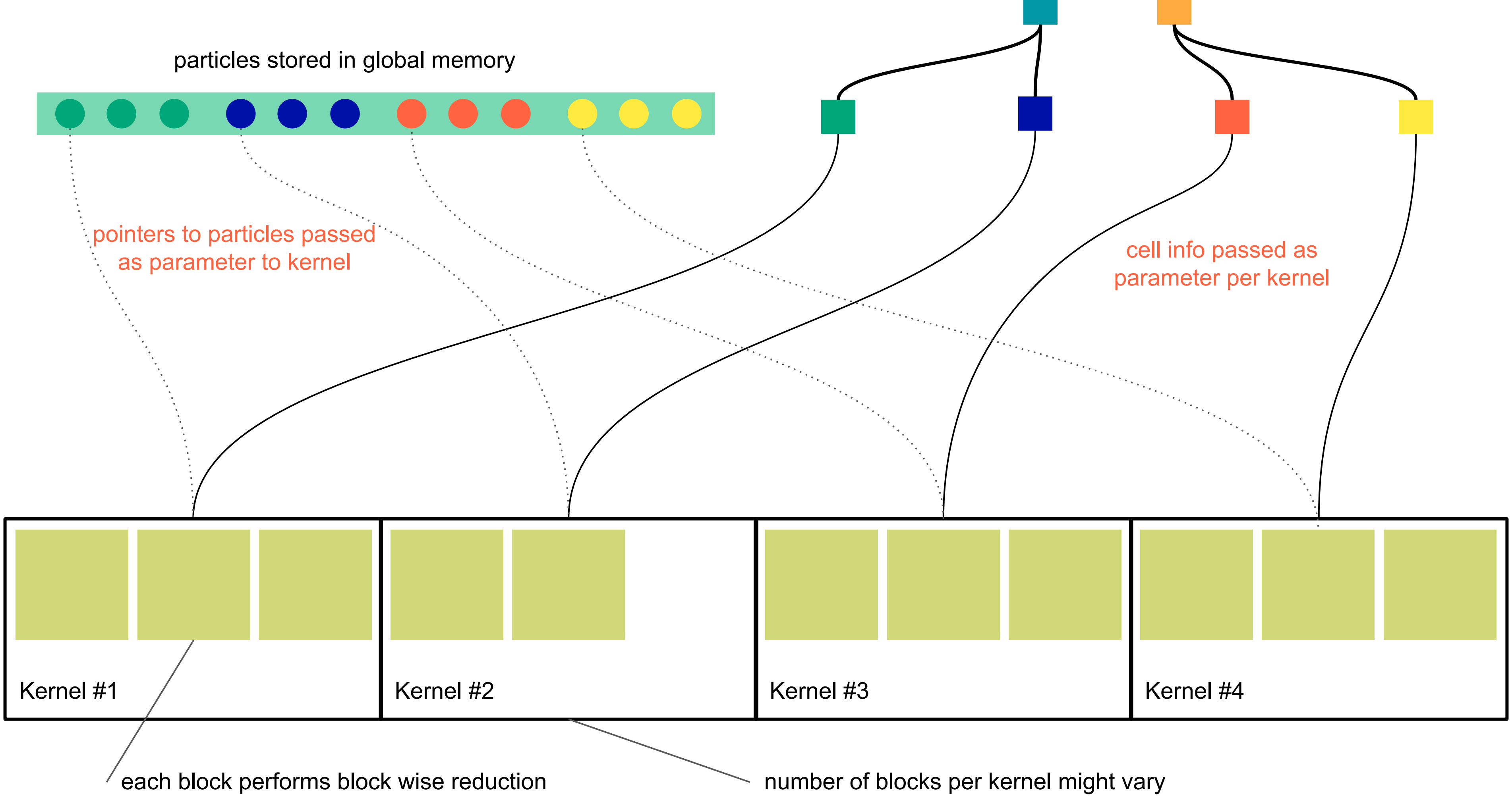

begin:end
index in memory & thread index
0:31
32:63
64:95
96:127
128:159
160:191
192:223
224:256
transfer from global to shared memory
0:31
32:63
64:95
96:127
128:159
160:191
192:223
224:256
reduce & synchronize
0:31
32:63
64:95
96:127
reduce & synchronize
0:31
32:63
reduce & synchronize
0:31
transfer from shared to local memory
...
...
warp level reduction
store result to global memory
local
shared
global
Not all the shared memory is really necessary

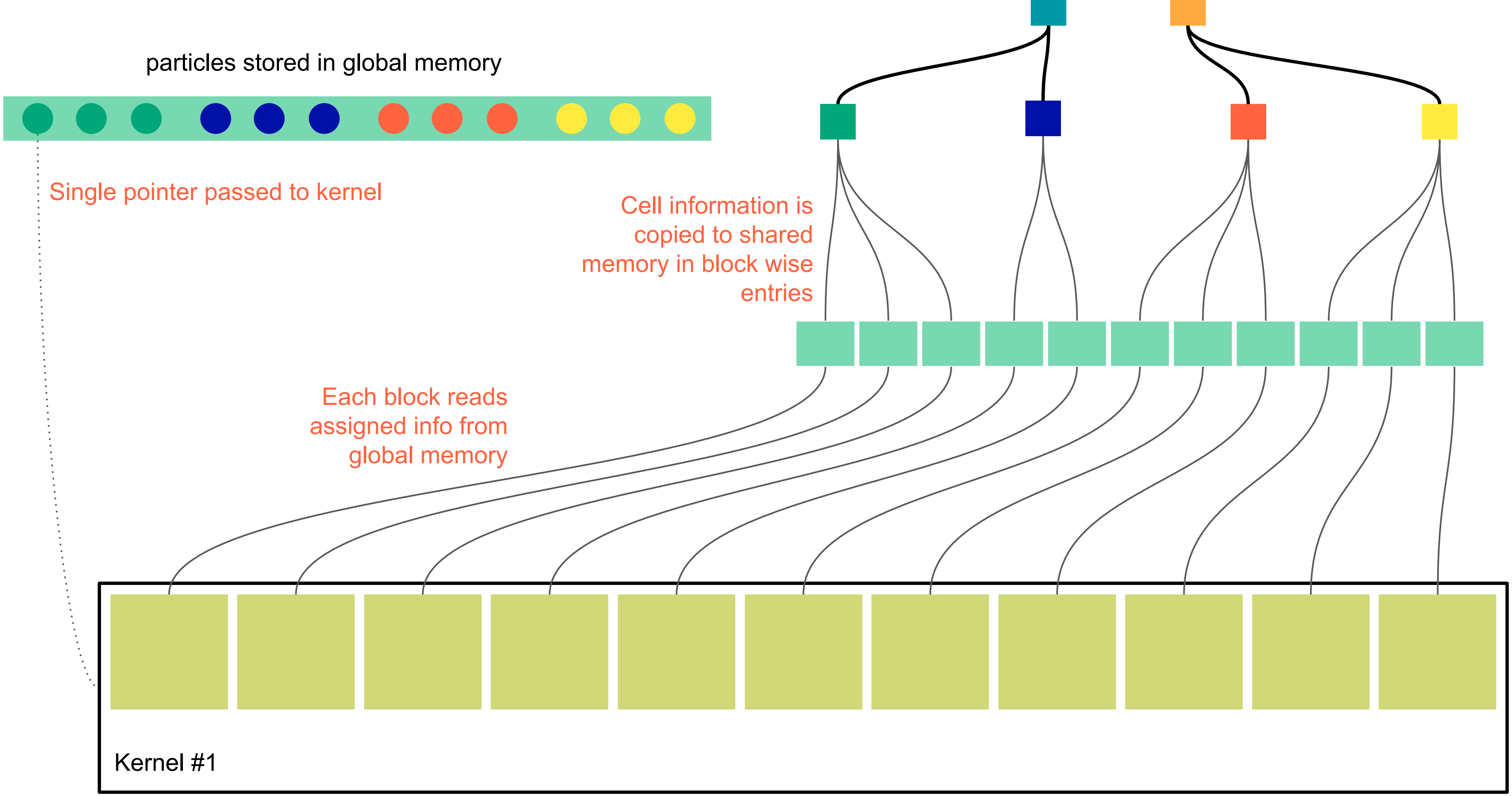
Bank conflict free access pattern



each thread of the same warp only accesses a single memory bank

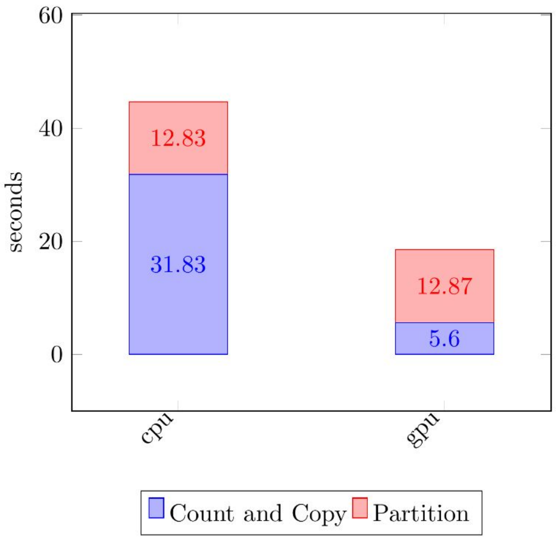

load all data from global to local memory
...
reductions on local memory using warp level primitives
store to shared memory with no bank conflicts
load to local memory
perform another warp level reduction
store result to global memory
local
shared
global


particles stored in global memory
pointers to particles passed as parameter to kernel
cell info passed as parameter per kernel
Kernel #1
Kernel #2
Kernel #3
Kernel #4
each block performs block wise reduction
number of blocks per kernel might vary


particles stored in global memory
Single pointer passed to kernel
Cell information is copied to shared memory in block wise entries
Each block reads assigned info from global memory
Kernel #1

Conclusion

- Successfully improved runtime of ORB 🎉
  - **5.6x** speedup. Theoretical limit was **6.2x**
  - Runtime also close to theory in absolute terms
- Better understanding of hardware
- Learned CUDA
- Improved C++ skills



Outlook

- ➔ Integrate to PKDGRAV
- ➔ Improve CPU partition
- ➔ Finish implementing GPU partition
  - ◆ not clear weather could give improvements
  - ◆ higher memory usage
- ➔ Data compression